

OVERVIEW **Creational Patterns III: Builder and Singleton**

After revising Abstract Factory and Factory Method, the lecture adds two creational patterns. Builder separates the step-by-step construction of a complex object from its representation: a Director calls the same build steps in order, and each ConcreteBuilder implements them differently (C++ vs. Java code, beef vs. chicken burger), returning the product at the end. Singleton ensures a class has only one instance with a global access point: private static instance, private constructor, public static getInstance(). A Prototype exercise (copying shapes with clone()) closes the creational patterns.

Builder roles

Director (runs the steps in order), Builder interface (buildPartA/B/C, getResult), ConcreteBuilders (one per representation), Product.

Builder vs. Abstract Factory

Builder builds one complex object step by step; Abstract Factory creates a family of related objects all at once.

Singleton recipe

Private constructor, private static instance field, public static getInstance() that creates the instance once and returns it.

Prototype

Objects copy themselves with clone() instead of being created with new.

HOW THE LECTURE IS ORGANIZED

Slides 2–6 Abstract Factory and Factory Method revised

Slides 7–21 Builder: intent, steps, structure, code generator, benefits, burger exercise

Slides 22–25 Builder: code-generator example in detail

Slides 26–37 Singleton: intent, structure, event manager, Java code, steps, benefits, banking logger

Slides 38–43 Prototype exercise and summary

EXAM FOCUS

Final 252 Q6 (5 marks) came straight from this lecture: Q6-1 (3 marks) asked for the Builder pattern and its class diagram for a sandwich-ordering system, and Q6-2 (2 marks) for the standard Singleton class diagram and its code.

HOW TO READ THIS PAGE

Every original slide is shown unchanged. **Yellow marks** on a slide point to a numbered note: hover or tap a mark to jump to its note, and hover a note to see its marks. Questions tagged **Past exam** appeared on a real paper; **Prediction** questions are likely angles, not confirmed.



Software Design and Architecture

[Creational Design Patterns] – Chapter 06, L03

NOTES

- 1 Last creational-patterns lecture: a revision of Abstract Factory and Factory Method, then **Builder** and **Singleton**. Both were on Final 252.

SLIDE 2 / 43 Lecture Outlines

Lecture Outlines

➤ Creational Patterns in Detailed Design

✓ Previously:

- Abstract Factory Pattern
- Factory Method Pattern

✓ **Builder**

- Code generator example

✓ **Singleton**

- Event manager example

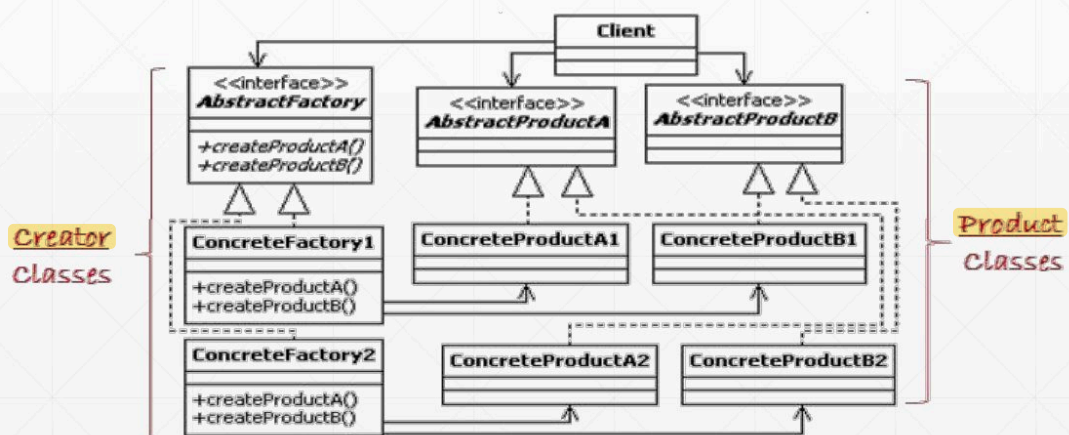
➤ What's next...

NOTES

- 1 Revision of the two factory patterns.
- 2 **Builder**, with a code-generator example.
- 3 **Singleton**, with an event-manager example.

SLIDE 3 / 43 Abstract Factory Pattern – Revised

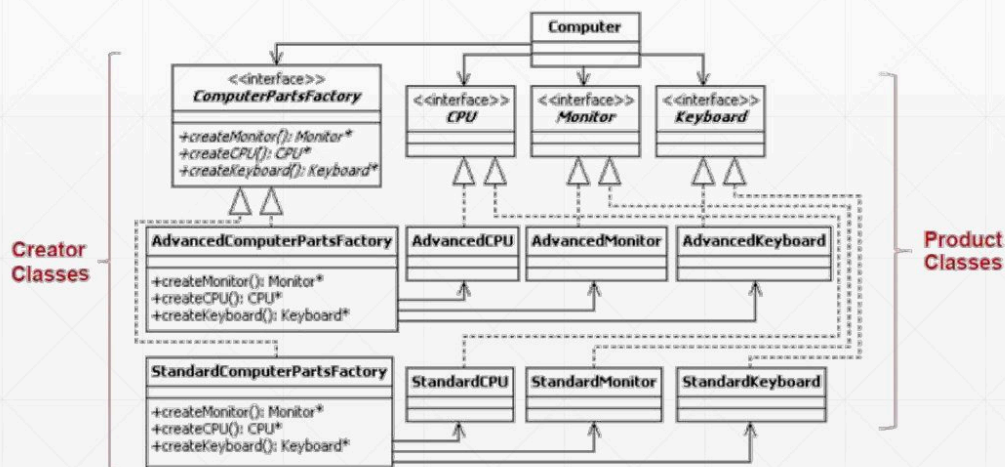
Abstract Factory Pattern – Revised



NOTES

- 1 Abstract Factory revised: an AbstractFactory with one create method per product; ConcreteFactory1 and 2 each create one **family** (A1+B1, A2+B2). Creator classes on the left, product classes on the right; the client uses only the interfaces.

Abstract Factory Pattern – Revised

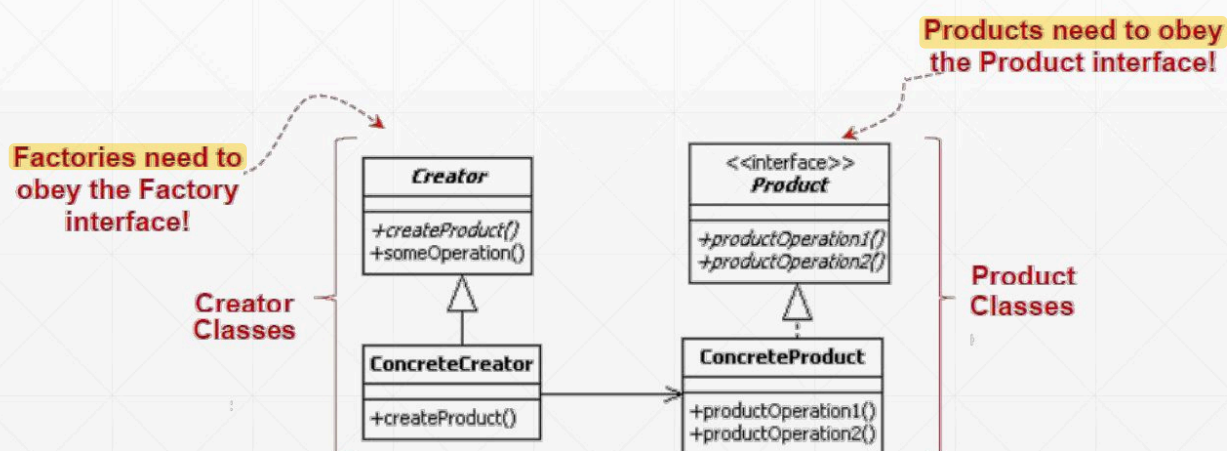


NOTES

- 1 The same structure applied to the computer store: ComputerPartsFactory creates a Monitor, CPU and Keyboard; the Advanced and Standard factories each create their own matching set.

SLIDE 5 / 43 Factory Method Pattern - Revised

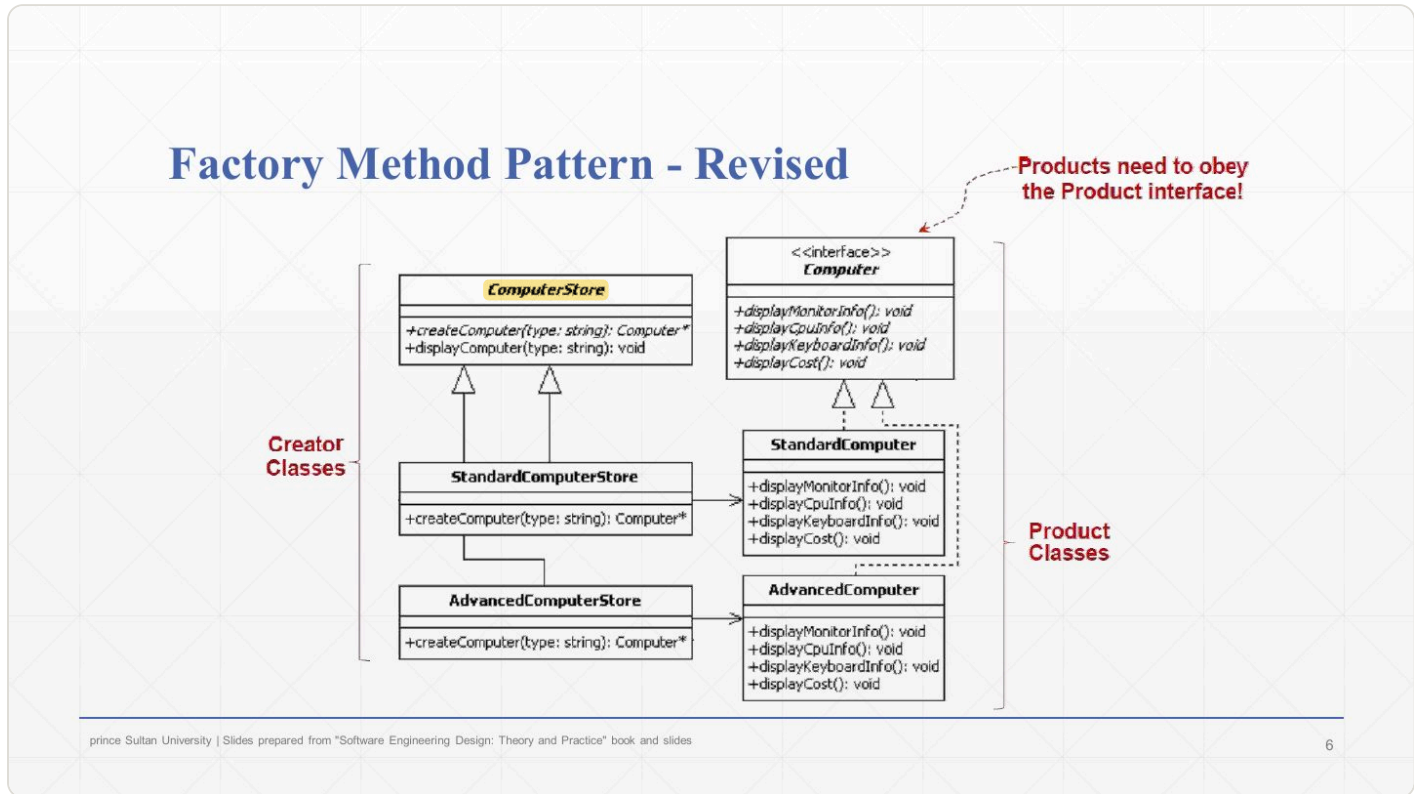
Factory Method Pattern - Revised



NOTES

- 1 Factory Method revised: an abstract **Creator** with *createProduct()* and *someOperation()*; a **ConcreteCreator** overrides *createProduct()* to return a **ConcreteProduct**. One product, chosen by a subclass. Factories must obey the factory (creator) interface; products must obey the product interface.

SLIDE 6 / 43 Factory Method Pattern – Revised: computer store



NOTES

- 1 Factory Method applied: ComputerStore's *createComputer(type)* is overridden by StandardComputerStore and AdvancedComputerStore, which return StandardComputer and AdvancedComputer through the Computer interface.

The Builder Pattern

- The **intent** of the Builder is to:
 - ✓ **Separate** the **construction** of a **complex object** from its **representation** so that the same construction process can create different representations.

NOTES

- 1 The **Builder's** intent: **separate the construction of a complex object from its representation...**
- 2 **...so that the same construction process can create different representations.** The steps (the algorithm) stay fixed; what each step produces varies with the builder.

EXAM ANGLE

Past exam · Final 252 · Q6-1 · 3 marks

Q1

Customized sandwiches must follow a correct, consistent assembly process; a Sandwich cannot be created until at least the bread and meat are set, and construction is separate from the Sandwich itself. Which pattern would you use? Draw its class diagram.

Answer: Pattern: Builder. It separates the construction of a complex object from its representation and builds it step by step. **SandwichBuilder** (interface): `setBread()`, `setMeat()`, `setCheese()`, `build()`. **CustomSandwichBuilder**: stores the parts; `build()` refuses to create the Sandwich unless bread and meat are set (consistency). **SandwichDirector**: runs the steps in the correct order. **Sandwich**: the product (bread, meat, cheese). Diagram: Director → (directs) SandwichBuilder; CustomSandwichBuilder → realizes SandwichBuilder and creates Sandwich.

Builder Design Pattern Example

- Steps for designing using the Builder pattern:
 - ✓ Identify and design the **product class** (e.g., **GeneratedProduct**)
 - ✓ Identify the product's **creational process and algorithm**, and design a class for its execution (e.g., **CodeGenerator**).
 - ✓ Using the knowledge acquired from steps 1 and 2, **design the builder interface**, which specifies the parts that need to be created for the whole product to exist. These are captured as abstract interface methods that need to be implemented by derived concrete builders.
 - ✓ Identify and design classes for the **different representations** of the product (e.g., CppMessageBuilder, JavaMessageBuilder, etc.)

NOTES

- 1 Step 1:** identify and design the **product** class (GeneratedProduct).
- 2 Step 2:** identify the product's **creational process and algorithm**, and design a class that executes it (CodeGenerator, the director).
- 3 Step 3:** from steps 1–2, design the **builder interface**: abstract methods for every part that must be created for the whole product to exist; concrete builders implement them.
- 4 Step 4:** design one class per **representation** of the product (CppMessageBuilder, JavaMessageBuilder...).

Extra:**Builder pattern**

- **Intent:** separate *how we build* a complex object (steps) from *what the final object looks like* (representation).
- **When useful:** same construction steps, but you want **different outputs** (e.g., generate Java vs. C# code; make small vs. large burger).
- **Key roles:**
 - **Director** — orchestrates steps in order (doesn't know details).
 - **Builder (interface)** — declares multistep methods (`buildPartA/B/C...`, `getProduct()`).
 - **Concrete Builders** — implement those steps to produce *different representations*.
 - **Product** — the thing being built.
- **Flow (always the same):**
 - Client chooses a `ConcreteBuilder` and gives it to the `Director`
 - `Director` runs the steps in sequence
 - Client asks builder for the final `Product`
- **Builder vs Abstract Factory :**
 - **Builder builds one complex object step-by-step.**
 - **Abstract Factory creates a family of related objects all at once.**

NOTES

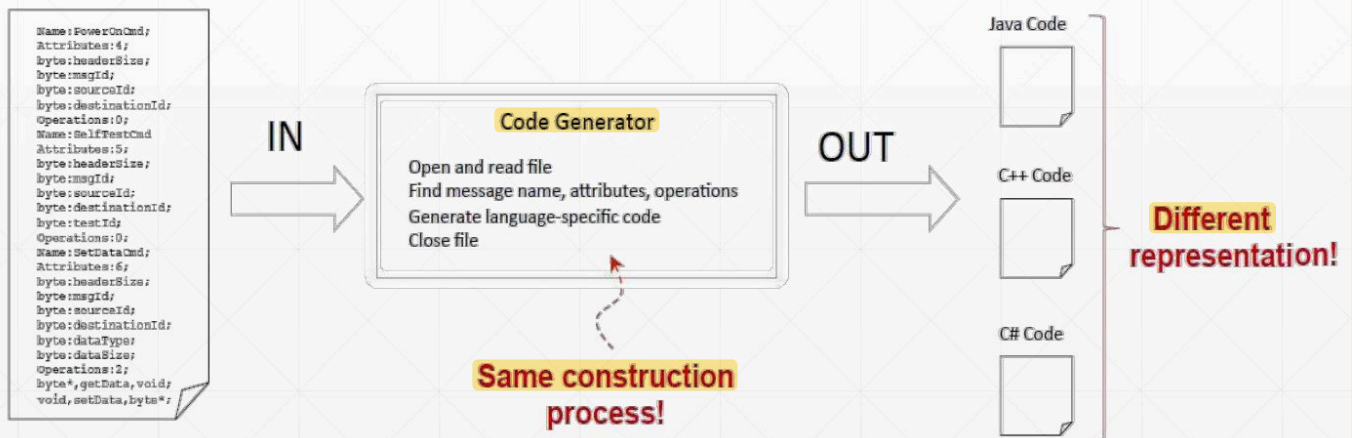
- 1 **Intent:** separate *how* a complex object is built (the steps) from *what* the result looks like (the representation).
- 2 **When useful:** same construction steps, different outputs, e.g. generating Java vs. C# code, or a small vs. large burger.
- 3 **Roles:** **Director** (orchestrates the steps in order, knows no details); **Builder** interface (declares the step methods `buildPartA/B/C...` and `getProduct()`); **ConcreteBuilders** (implement the steps for each representation); **Product** (the thing built).
- 4 **Flow:** the client chooses a `ConcreteBuilder` and gives it to the `Director` → the `Director` runs the steps in sequence → the client asks the builder for the finished product.
- 5 **Builder vs. Abstract Factory:** Builder builds **one complex object step by step**; Abstract Factory creates **a family of related objects all at once**.

EXAM ANGLE**Prediction****Q1**

Contrast the Builder and Abstract Factory patterns.

Answer: Builder constructs one complex object step by step under a director's control, returning it at the end; different builders give different representations of the same object. Abstract Factory creates a family of related objects, each in a single call, ensuring the family is consistent.

Builder Design Pattern Example

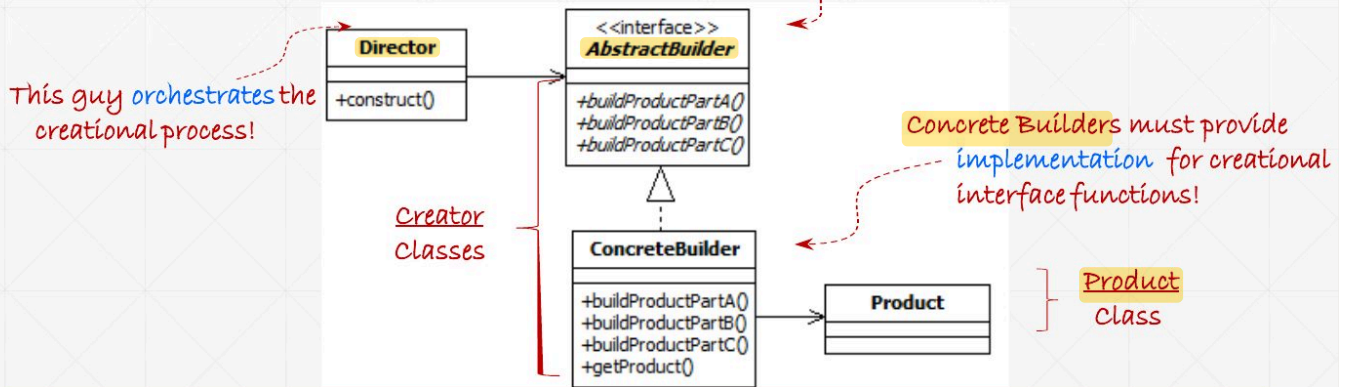


NOTES

- 1 The code-generator example as a picture. **In:** a messaging-specification document (message names, attributes, operations).
- 2 The **code generator** runs the **same construction process** for every language: open and read the file, find message names, attributes and operations, generate language-specific code, close the file.
- 3 **Out:** Java, C++ or C# code, i.e. **different representations** of the same messages. That combination (fixed process, varying output) is exactly the Builder's use case.

The Builder Pattern

➤ Builder Structure



prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

11

NOTES

- 1 **Director** with `construct()`: "this guy orchestrates the creational process". It holds an `AbstractBuilder` and calls its steps in order.
- 2 «interface» **AbstractBuilder** with `buildProductPartA/B/C()`: these methods specify the multistep creational process for the product.
- 3 **ConcreteBuilder** must implement those creational methods, plus `getProduct()`.
- 4 **Product**: the class being built; the `ConcreteBuilder` creates and assembles it. Creator classes = `Director`, `AbstractBuilder`, `ConcreteBuilder`; product class = `Product`.

EXAM ANGLE

Prediction

Q1

In the Builder pattern, which participant calls the construction steps in the right order?

- A. Director
- B. ConcreteBuilder
- C. Product
- D. Client

Answer: A. Director

Extra:**➤ Builder Pattern — How to Solve (General Steps)****➤ 0) Recognize it's a Builder problem**

- ✓ Same **sequence of steps**, different **final representations** (outputs).
- ✓ Need to **construct a complex object step-by-step**.
- ✓ You want to **hide construction** from the client and make adding new variants easy.

➤ 1) Define the Product

- ✓ Name the final thing being built (e.g., Report, Burger, Form, Query, House).
- ✓ List its parts/fields (what varies between variants).

➤ 2) Declare the Builder interface (steps)

- ✓ Write the **multistep API** that builds parts of the Product.

Typical methods:

```
buildPartA(...)
buildPartB(...)
buildPart.....
getResult() (returns the Product)
```

Steps should describe *what to build*, not how.**3) Implement Concrete Builders (variants)**

- ✓ One class per variant (e.g., PdfReportBuilder, HtmlReportBuilder).
- ✓ Each implements the steps **its own way** and stores intermediate state.
- ✓ getResult() returns the fully built Product.

4) Create the Director (optional but recommended)

- ✓ Orchestrates **order** of steps.
- ✓ Has a method like construct() that calls the steps in sequence.
- ✓ Keeps the construction process reusable and consistent.

5) Client wiring

- ✓ Client picks the **ConcreteBuilder** (the variant desired).
- ✓ Pass it to the **Director**, call construct().
- ✓ Get the Product via builder.getResult() (or director.construct() returns it).

6) Verify extensibility

- ✓ To add a new variant, create a **new ConcreteBuilder**; only—client and director stay the same.

NOTES

- 0) Recognise a Builder problem:** the same sequence of steps must give different final representations; a complex object is built step by step; construction should be hidden from the client and new variants easy to add.
- 1) Define the product:** name what is built (Report, Burger, Form, House) and list its parts.
- 2) Declare the builder interface:** the multistep API (`buildPartA(...)`, `buildPartB(...)`, `getResult()`). Steps say *what* to build, not how.
- 3) Implement concrete builders:** one class per variant (PdfReportBuilder, HtmlReportBuilder), each storing intermediate state; `getResult()` returns the finished product.
- 4) Create the director** (optional but recommended): a `construct()` method calls the steps in sequence, keeping the process reusable and consistent. **5) Client wiring:** pick a ConcreteBuilder, pass it to the Director, call `construct()`, get the product.
- 6) Verify extensibility:** a new variant needs only a new ConcreteBuilder; client and director stay unchanged.

1. Product

```

java

class Product {
    // the final object being built
}

```

2. Builder Interface

```

java

interface Builder {
    void buildPartA();
    void buildPartB();
    Product getResult();
}

```

•Points to Remember

- Product** = the final object.
- Builder interface** = declares building steps.
- Concrete Builders** = implement steps differently (Java vs. C#).
- Director** = controls the order of building steps.
- Client** = chooses which builder to use.

🔗 Same **construction steps**, but **different final representations**.

3. Concrete Builders

```

java

class JavaBuilder implements Builder {
    public void buildPartA() { /* build step A for Java */ }
    public void buildPartB() { /* build step B for Java */ }
    public Product getResult() { return new Product(); }
}

class CSharpBuilder implements Builder {
    public void buildPartA() { /* build step A for C# */ }
    public void buildPartB() { /* build step B for C# */ }
    public Product getResult() { return new Product(); }
}

```

4. Director

```

java

class Director {
    private Builder builder;

    Director(Builder builder) {
        this.builder = builder;
    }

    Product construct() {
        builder.buildPartA();
        builder.buildPartB();
        return builder.getResult();
    }
}

```

5. Client Code

```

java

public class Demo {
    public static void main(String[] args) {
        // Use Java builder
        Builder javaBuilder = new JavaBuilder();
        Director director = new Director(javaBuilder);
        Product javaProduct = director.construct();

        // Use C# builder
        Builder csBuilder = new CSharpBuilder();
        Director director2 = new Director(csBuilder);
        Product csProduct = director2.construct();
    }
}

```

NOTES

1 The skeleton in Java:

CODE

WHAT IT DOES

```
class Product { /* the final object
being built */ }
```

1. The product.

```
interface Builder { void buildPartA();
void buildPartB(); Product
getResult(); }
```

2. The builder interface.

```
class JavaBuilder implements Builder {
buildPartA() { /* step A for Java */ }
... getResult() { return new Product();
} }
```

3. Concrete builders (CSharpBuilder the same, with C# steps).

```
class Director { private Builder
builder; Director(Builder builder) {
this.builder = builder; }
```

4. The director receives a builder...

```
Product construct() {
builder.buildPartA();
builder.buildPartB(); return
builder.getResult(); }
```

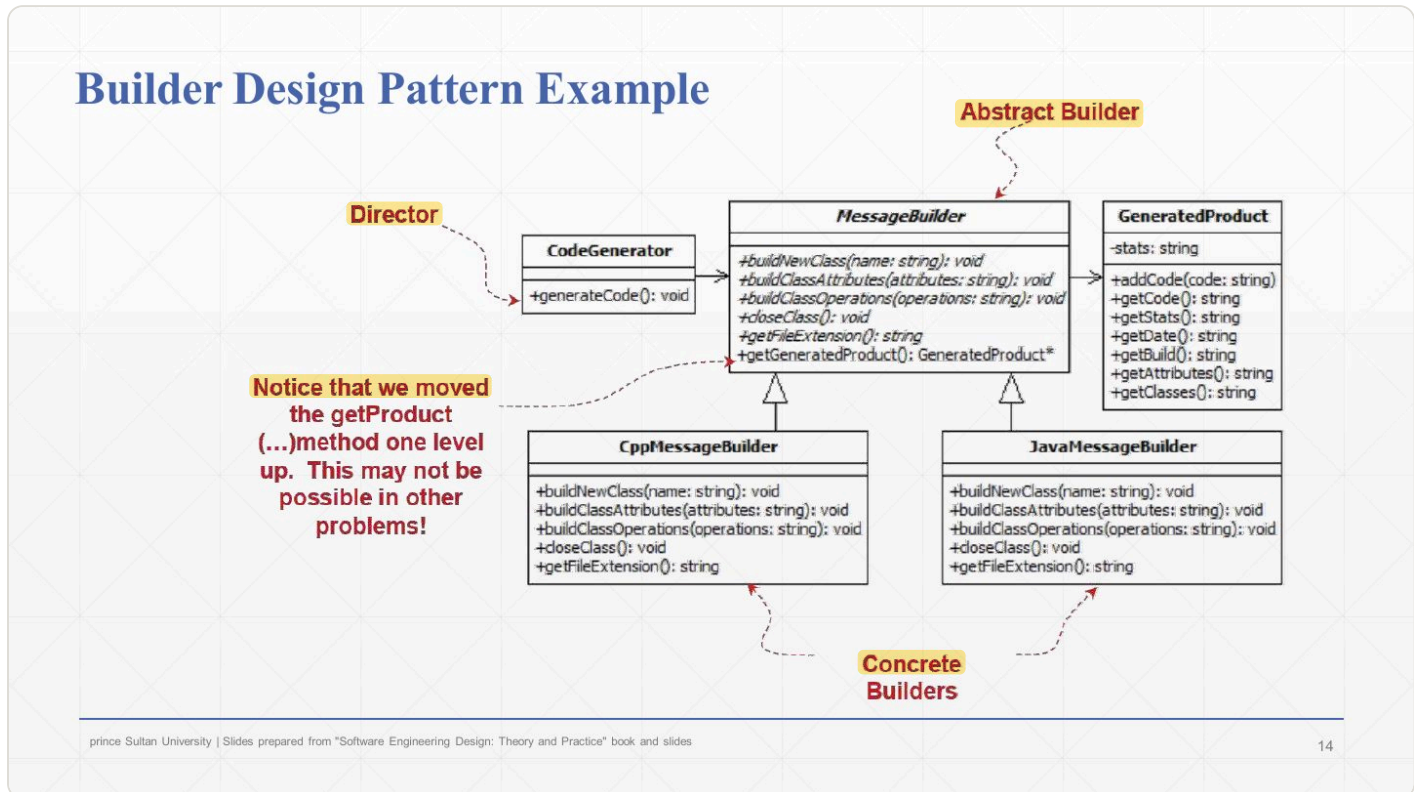
...and runs the steps in a fixed order.

```
Builder javaBuilder = new
JavaBuilder(); Director director = new
Director(javaBuilder); Product
javaProduct = director.construct();
```

5. The client chooses the builder; C# is the same three lines with CSharpBuilder.

- 2 **Points to remember:** product = the final object; builder interface = declares the steps; concrete builders = implement them differently; director = controls the order; client = chooses the builder. Same construction steps, different final representations.

SLIDE 14 / 43 **Builder Design Pattern Example: class diagram**



NOTES

- 1 The code-generator design. **Director:** `CodeGenerator` (`generateCode()`).
- 2 **Abstract builder:** `MessageBuilder` with the steps `buildNewClass(name)`, `buildClassAttributes(attributes)`, `buildClassOperations(operations)`, `closeClass()`, `getFileExtension()`, plus `getGeneratedProduct()`.
- 3 **Concrete builders:** `CppMessageBuilder` and `JavaMessageBuilder` implement every step for their language. The **product** is `GeneratedProduct` (`addCode()`, `getCode()`, `getStats()` ...).
- 4 Note: `getGeneratedProduct()` was moved up into the abstract builder, because every language returns the same product type. That isn't possible in every problem (if products differ per builder, each builder returns its own).

The Builder provides the following benefits

- **Separates** an object's construction process from its representation; therefore future representations can be added easily to the software.
- **Changes to existing representations** can be made without modifying the code for the creational process.
- Provides **finer control** over the construction process so that parts of objects can be created at **discrete points** in time (which differs from the Abstract Factory pattern)
- Improves **maintainability, testability, and reusability**.

NOTES

- 1 It **separates** construction from representation, so new representations (a C# builder) are added easily.
- 2 **Existing representations change** without touching the creational process (the director).
- 3 It gives **finer control** over construction: parts of an object are created at **discrete points in time**, unlike Abstract Factory, which creates each product in one call.
- 4 It improves **maintainability, testability and reusability**.

EXAM ANGLE

Prediction

Q1

Give two benefits of the Builder pattern.

Answer: Construction is separated from representation, so new representations can be added without changing the construction process; it gives finer control, building parts at discrete points; it improves maintainability, testability and reuse.

Extra:♦ **Idea in Simple Words**

The **Builder Pattern** is used when you want to create a complex object **step by step**, instead of building it all at once.

🍔 Think of it like ordering food:

- You don't get the meal instantly
- You choose parts (bread, meat, sauce...)
- Then it gets assembled step by step

♦ **Real-Life Example: 🍔 Burger Builder**

Imagine you are building a burger:

✗ **Without Builder (hard way)**

You create everything at once:

- `Burger("big bun", "double meat", "cheese", "ketchup", "lettuce")`

📋 **Problem:**

- Hard to read
- Hard to customize
- Easy to make mistakes

✔ **With Builder Pattern**

You build it step by step:

```
java
Burger burger = new BurgerBuilder()
    .addBun("Big Bun")
    .addPatty("Beef")
    .addCheese()
    .addLettuce()
    .addSauce("Ketchup")
    .build();
```

NOTES

- 1 In simple words: use Builder when an object is created step by step instead of all at once, like ordering food where you pick bread, meat and sauce and the meal is assembled.
- 2 Without it: `Burger("big bun", "double meat", "cheese", "ketchup", "lettuce")`, a constructor with many parameters that is hard to read, hard to customise and easy to get wrong.
- 3 With it: `new BurgerBuilder().addBun("Big Bun").addPatty("Beef").addCheese().addLettuce().addSauce("Ketchup").build();`, each step named and optional. (This fluent, chained style is a common variant of Builder without a director.)

Question:

A restaurant's system needs to prepare different types of burgers such as **Beef Burger** and **Chicken Burger**. Each burger must always be prepared in the same sequence of steps:

1. **Add Bun**
2. Add Patty
3. Add Sauce

The ingredients, however, differ depending on the type of burger:

- **Beef Burger:** **Sesame Bun** + Beef Patty + BBQ Sauce
- **Chicken Burger:** Whole Wheat Bun + Chicken Patty + Mayo Sauce

You are asked to **design and implement** this system in Java such that:

- The **construction steps are the same** for all burgers.
- New types of burgers can be added easily in the future.
- The client should only decide the burger type, while the actual construction process remains hidden.

Tasks: 1- **draw class diagram based on builder pattern**

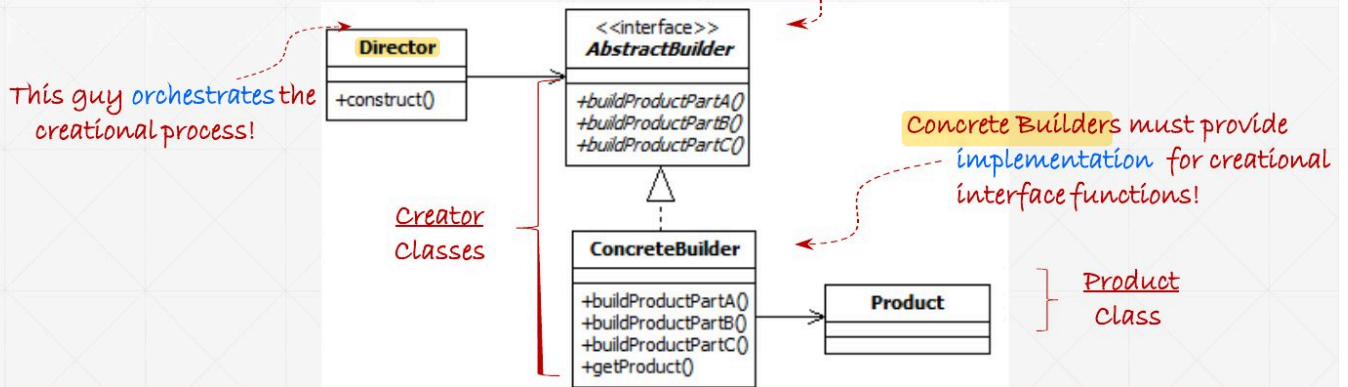
2- **write the java code**

NOTES

- 1 The exercise: a restaurant prepares **Beef** and **Chicken** burgers.
- 2 Every burger is prepared in the **same sequence**: add bun, add patty, add sauce.
- 3 The **ingredients differ**: beef = sesame bun + beef patty + BBQ sauce; chicken = whole-wheat bun + chicken patty + mayo.
- 4 Requirements: same steps for all burgers, new burger types easy to add, and the client chooses only the type while construction stays hidden: textbook Builder.
- 5 Tasks: draw the class diagram and write the Java (solution on slides 19–20).

The Builder Pattern

➤ Builder Structure



prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

18

NOTES

- 1 The structure again, as the template for the burger answer: Director (Chef) → AbstractBuilder (BurgerBuilder) ← ConcreteBuilders (BeefBurgerBuilder, ChickenBurgerBuilder) → Product (Burger).

SLIDE 19 / 43 Burger builder: solution code

1. Product (Burger)

```

java
class Burger {
    String bun;
    String patty;
    String sauce;

    public String toString() {
        return bun + " " + patty + " " + sauce;
    }
}
  
```

2. Builder Interface

```

java
interface BurgerBuilder {
    void addBun();
    void addPatty();
    void addSauce();
    Burger getBurger();
}
  
```

3. Concrete Builders (Different Recipes)

```

java
class BeefBurgerBuilder implements BurgerBuilder {
    Burger burger = new Burger();

    public void addBun() { burger.bun = "Sesame Bun"; }
    public void addPatty() { burger.patty = "Beef Patty"; }
    public void addSauce() { burger.sauce = "BBQ Sauce"; }
    public Burger getBurger() { return burger; }
}

class ChickenBurgerBuilder implements BurgerBuilder {
    Burger burger = new Burger();

    public void addBun() { burger.bun = "Whole Wheat Bun"; }
    public void addPatty() { burger.patty = "Chicken Patty"; }
    public void addSauce() { burger.sauce = "Mayo Sauce"; }
    public Burger getBurger() { return burger; }
}
  
```

4. Director

```

java
class Chef {
    private BurgerBuilder builder;

    Chef(BurgerBuilder builder) {
        this.builder = builder;
    }

    // construct() = step-by-step process
    public Burger construct() {
        builder.addBun();
        builder.addPatty();
        builder.addSauce();
        return builder.getBurger();
    }
}
  
```

19

NOTES

1 The solution:

CODE	WHAT IT DOES
<pre>class Burger { String bun; String patty; String sauce; public String toString() { return bun + " + " + patty + " + " + sauce; } }</pre>	1. The product.
<pre>interface BurgerBuilder { void addBun(); void addPatty(); void addSauce(); Burger getBurger(); }</pre>	2. The three steps plus the result.
<pre>class BeefBurgerBuilder implements BurgerBuilder { Burger burger = new Burger();</pre>	3. The beef recipe keeps its burger under construction...
<pre> public void addBun() { burger.bun = "Sesame Bun"; } ... addPatty → "Beef Patty" ... addSauce → "BBQ Sauce"</pre>	...and fills each part its own way.
<pre>class ChickenBurgerBuilder implements BurgerBuilder { ... "Whole Wheat Bun" ... "Chicken Patty" ... "Mayo Sauce" }</pre>	The chicken recipe: same steps, other ingredients.
<pre>class Chef { private BurgerBuilder builder; Chef(BurgerBuilder builder) { this.builder = builder; }</pre>	4. The director.
<pre> public Burger construct() { builder.addBun(); builder.addPatty(); builder.addSauce(); return builder.getBurger(); } }</pre>	The fixed order lives here, once.

5. Client Code

```

java

public class Demo {
    public static void main(String[] args) {
        Chef beefChef = new Chef(new BeefBurgerBuilder());
        Burger beefBurger = beefChef.construct();
        System.out.println("Built: " + beefBurger);

        Chef chickenChef = new Chef(new ChickenBurgerBuilder());
        Burger chickenBurger = chickenChef.construct();
        System.out.println("Built: " + chickenBurger);
    }
}

```

NOTES

1 5. The client:

CODE

```
Chef beefChef = new Chef(new
BeefBurgerBuilder());
```

WHAT IT DOES

Choose the burger type by choosing the builder.

```
Burger beefBurger =
beefChef.construct();
System.out.println("Built: " +
beefBurger);
```

Prints "Built: Sesame Bun + Beef Patty + BBQ Sauce".

```
Chef chickenChef = new Chef(new
ChickenBurgerBuilder()); Burger
chickenBurger =
chickenChef.construct();
```

Same director code, different representation.

2 A Veggie burger would need only a VeggieBurgerBuilder; Chef and the client logic stay unchanged.

Builder Design Pattern

- The Builder Design pattern is an object **creational pattern** that **encapsulates both the creational process and the representation of product objects.**
- In the **Abstract Factory** Pattern, various product objects are **created** all at once,
- The **Builder** Pattern allows clients to control the **(multistep)** creational process of a **single product object**.
- The Builder dictates the creation of **individual parts** of the object at **discrete points** throughout software operations.

NOTES

- 1 Builder is an object creational pattern that **encapsulates both the creational process and the representation** of product objects.
- 2 In **Abstract Factory**, the various product objects are **created all at once** (one call each).
- 3 Builder lets clients control the **multistep** creational process of a **single** product.
- 4 It dictates the creation of individual parts at **discrete points** throughout the software's operation.

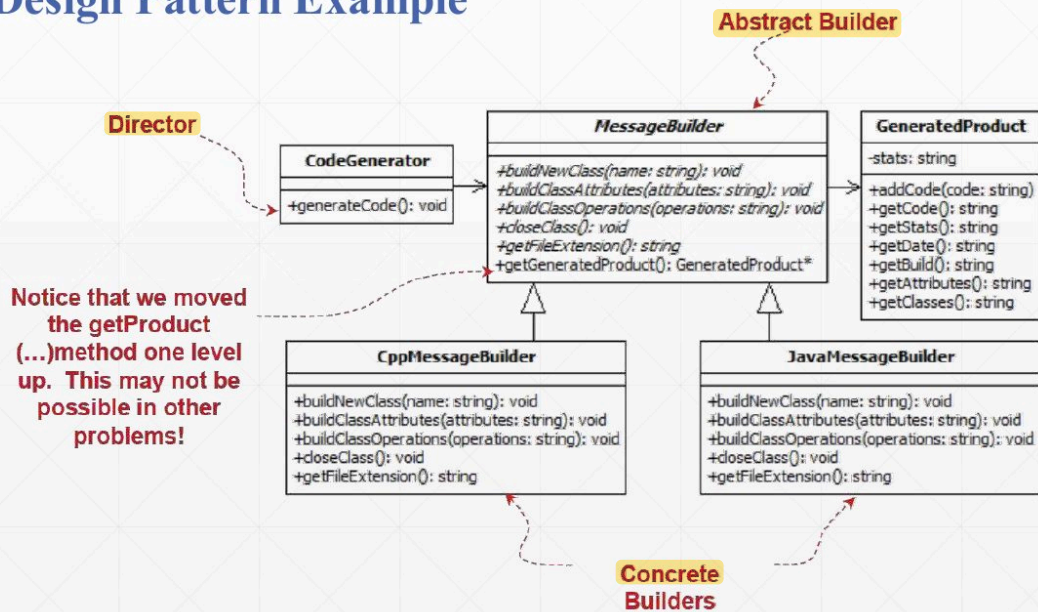
Builder Design Pattern Example

- Consider a **code generator**:
 - ✓ Code generated from an **interface specification document (ICD)** used to monitor and control custom developed hardware.
 - ✓ Company sells hardware, but also includes library to monitor and control its hardware.
 - ✓ Generator reads a text document containing information about a messaging specification. It then **generates C++ classes** that can be used to monitor and control hardware. This is also shipped to customers so that they can use the hardware easily.
 - ✓ The ICD is quite complex, and customers are demanding support for other languages, such as **Java and C#**.

NOTES

- 1 The textbook example: a **code generator** works from an interface specification document (ICD) for custom hardware.
- 2 The company sells the hardware plus a library to monitor and control it. The generator reads the messaging specification and **generates C++ classes** that customers use with the hardware.
- 3 The ICD is complex, and customers now want **Java and C#** as well: the same generation process, new representations, so Builder.

Builder Design Pattern Example



prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

23

NOTES

- The design (same as slide 14): CodeGenerator is the **director**, MessageBuilder the **abstract builder**, CppMessageBuilder and JavaMessageBuilder the **concrete builders**, GeneratedProduct the **product**. A C# version would add only a CSharpMessageBuilder.

Builder Design Pattern Example

The same steps used to create products in all target languages!

```

// Forward reference.
class GeneratedProduct;

class MessageBuilder {
public:
    // The interface method for building a new class.
    virtual void buildNewClass(string name) = 0;

    // The interface method for building class attributes.
    virtual void buildClassAttributes(string attributeList) = 0;

    // The interface method for building class operations.
    virtual void buildClassOperations(string operationList) = 0;

    // The interface method for closing a new class.
    virtual void closeClass() = 0;

    // The file extension for the target programming language.
    virtual string getFileExtension() = 0;

    // Return the generated product.
    GeneratedProduct* getGeneratedProduct() {
        return _codeProduct;
    }

private:
    // The product containing generated code and stats about code generated.
    GeneratedProduct* _codeProduct;
};
  
```

The Builder Interface for generating code!

prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

24

NOTES

1 The builder interface in C++:

CODE	WHAT IT DOES
<pre>class MessageBuilder { public:</pre>	The builder interface for generating code.
<pre> virtual void buildNewClass(string name) = 0;</pre>	Step: start a class. <code>= 0</code> makes it pure virtual (abstract).
<pre> virtual void buildClassAttributes(string attributeList) = 0;</pre>	Step: its attributes.
<pre> virtual void buildClassOperations(string operationList) = 0;</pre>	Step: its operations.
<pre> virtual void closeClass() = 0;</pre> <pre> virtual string getFileExtension() = 0;</pre>	Step: close it; and the language's file extension.
<pre> GeneratedProduct* getGeneratedProduct() { return _codeProduct; }</pre>	Shared by all builders: return the product.
<pre>private: GeneratedProduct* _codeProduct; };</pre>	The product holds the generated code and statistics.

2 The same steps are used to create products in every target language.

SLIDE 25 / 43 Builder Design Pattern Example: the C++ builder

Builder Design Pattern Example

```

class CppMessageBuilder : public MessageBuilder {
public:
    // The interface method for building a new class.
    virtual void buildNewClass(string name) {
        // Generate code for creating a class using CPP style and the name
        // argument.
        // Once code is generated, add it to the product.
        getGeneratedProduct()->addCode(/*new C++ class code*/);
    }

    // The interface method for building class attributes.
    virtual void buildClassAttributes(string attributeList) {
        // For all items in attributeList, generate attributes using CPP style
        // and add them to the generated code.
        // Once code is generated, add it to the product.
        getGeneratedProduct()->addCode(/*C++ attributes*/);
    }

    // The interface method for building class operations.
    virtual void buildClassOperations(string operationList) {
        // For all items in operationList, generate operations using CPP style
        // and add them to the generated code.
        // Once code is generated, add it to the product.
        getGeneratedProduct()->addCode(/*C++ operations*/);
    }

    // The interface method for closing a new class.
    virtual void closeClass() {
        // Generate code to close a class in Cpp, and add it to the generated
        // code.
        // Once code is generated, add it to the product.
        getGeneratedProduct()->addCode("\n");
    }
}

```

C++ Builder

Step 1

```
class MessageOne
{
```

Step 2

```
class MessageOne
{
private:
    int attributeOne;
    int attributeTwo;

```

Step 3

```
class MessageOne
{
private:
    int attributeOne;
    int attributeTwo;

public:
    void setValueOne(unsigned char x);

```

Step 4

```
class MessageOne
{
private:
    int attributeOne;
    int attributeTwo;

public:
    void setValueOne(unsigned char x);
};

```

C++ Product Created!

prince Sultan University | Slides prepared for

NOTES

- 1 The C++ concrete builder, step by step (each step adds its code to the product with `getGeneratedProduct()->addCode(...)`).
- 2 **Step 1** `buildNewClass(name)` writes `class MessageOne {`. **Step 2** `buildClassAttributes` adds `private: int attributeOne; int attributeTwo;`. **Step 3** `buildClassOperations` adds `public: void setValueOne(unsigned char x);`. **Step 4** `closeClass()` writes `};`.
- 3 Result: a complete C++ class, the "C++ product created". `JavaMessageBuilder` runs the same four calls and produces a Java class instead.

SLIDE 26 / 43 Singleton Design Pattern

Singleton Design Pattern

- The Singleton design pattern is an **object creational** design pattern used to **prevent objects** from being instantiated more than once in a running program.
- The **intent** of the Singleton is to
 - ✓ **Ensure a class** only has **one instance**, and provide a **global** point of access to it.

NOTES

- 1 **Singleton** is an object creational pattern that **prevents a class from being instantiated more than once** in a running program.
- 2 Its intent: **ensure a class has only one instance, and provide a global point of access to it.**

Prediction

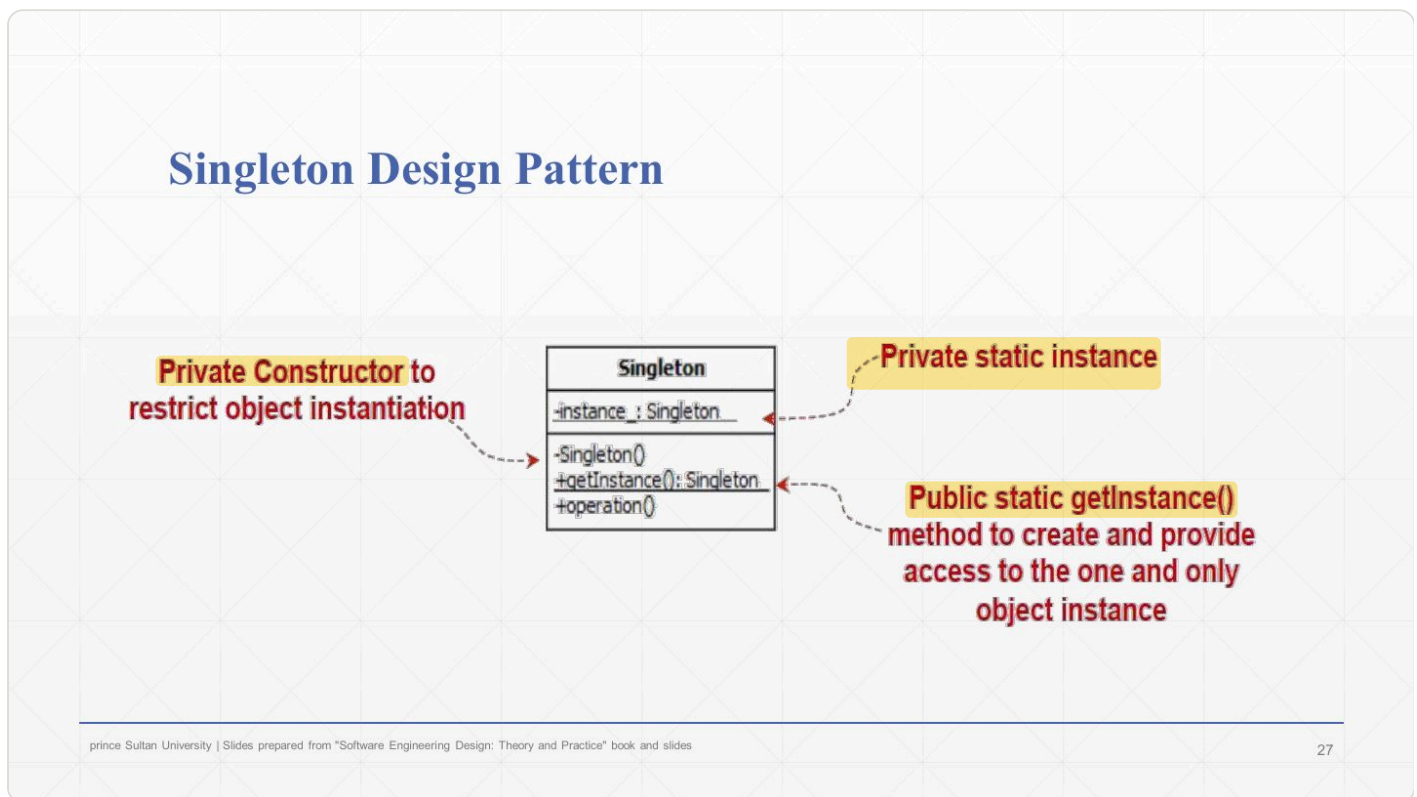
Q1

Which pattern guarantees exactly one instance of a class with a global access point?

- A. Singleton
- B. Prototype
- C. Factory Method
- D. Builder

Answer: A. Singleton

SLIDE 27 / 43 Singleton Design Pattern



NOTES

- 1 The UML, member by member. `-Singleton()`: a **private constructor** (minus sign) restricts instantiation, so no other class can call `new Singleton()`.
- 2 `-instance : Singleton` (underlined): a **private static** attribute that holds the one instance. Underlined members are static in UML.
- 3 `+getInstance() : Singleton` (underlined): a **public static** method that creates the instance if needed and gives everyone access to the one and only object. `+operation()` stands for the class's ordinary behaviour.

Past exam · Final 252 · Q6-2 · 2 marks

Q1

Draw the standard class diagram of the Singleton pattern and convert it into code (any language).

Answer: Diagram: class **Singleton** with - instance: Singleton {static}, - Singleton(), + getInstance(): Singleton {static}. Code:

```
public class Singleton { private static Singleton instance; private Singleton() { } public static Singleton getInstance() { if (instance == null) { instance = new Singleton(); } return instance; } }
```

. The private constructor stops other classes calling `new`; the instance is created on first use and the same one is returned afterwards.

SLIDE 28 / 43 Singleton Design Pattern Example

Singleton Design Pattern Example

Problem:

Consider an application that requires **event logging** capabilities. The application consists of many different objects that generate events to keep track of their actions, status of operations, errors, or any other information of interest. A decision is made to **create an event manager** that can be accessed by all objects and used to **manage all events in the system**. Upon instantiation, the **event manager** creates an **event list** that gets updated as events are logged. At specific points during the software system's operation, these events are written to a file. To prevent conflicts, it is desirable that at any given time, there is **only one instance of the event manager** executing.

Extra:

- ◆ The application needs **event logging** (recording actions, statuses, and errors).
- ◆ Many objects in the system generate events, but instead of each one handling logging separately, a **central Event Manager** is created.
- ◆ This Event Manager keeps an **event list**, updates it when events happen, and later writes the events to a file.
- ◆ To avoid problems (like multiple versions writing at the same time), the design ensures there is **only one Event Manager instance** in the whole system.

NOTES

- 1 The problem: an application needs **event logging**. Many objects generate events (actions, status, errors).
- 2 The design: one central **event manager** that every object uses. It creates an event list on instantiation, adds events as they are logged, and writes them to a file at specific points.
- 3 To prevent conflicts (two managers writing the same file), there must be **only one instance of the event manager** at any time: Singleton.

Extra:**Why Is There a Problem?**

In many software systems:

- Multiple **parts of the application** (e.g. login system, payment system, notification system) need to **log events** — such as user actions, errors, system updates.
- If each part **creates its own logger**, this leads to:
 - ✗ **Multiple log files** or conflicting writes.
 - ✗ **Confusing or inconsistent data.**
 - ✗ **Harder to debug** or audit the system.
- **The real problem:** If every part creates a new logger (event manager), it causes **conflicts, redundancy, and loss of control.**

Why Singleton Fixes This?

- Singleton ensures that **only one instance** of the **EventManager (Logger)** exists.
 - All parts of the system **share the same logger.**
 - You maintain **one log file**, centralized control, and avoid conflicts.
- ✓ One object → One responsibility → No conflict.

NOTES

- 1 Why there is a problem: login, payment and notification systems all need to log. If each creates its own logger, you get **multiple log files or conflicting writes**, inconsistent data, and a system that is harder to debug and audit.
- 2 The real problem is conflicts, redundancy and loss of control.
- 3 Singleton fixes it: only one EventManager exists, every part shares it, so there is one log file and central control. One object → one responsibility → no conflict.

Singleton Design Pattern Example



NOTES

- 1 The example mapped onto the generic structure: **EventManager** has `-instance_ : EventManager*` (static), a private constructor `-EventManager()`, `+getInstance(): EventManager*` (static) and `+logEvent(eventDescription: string)`.
- 2 Compared with the generic **Singleton** box, only the names changed: `logEvent` is the class's real `operation()`.

Java Code for the Singleton Pattern

A **static** variable is a variable that has been allocated "statically", meaning that its **lifetime** (or "extent") is the **entire run** of the program.

The Singleton class

```
// File Name: Singleton.java
public class Singleton {

    private static Singleton singleton = new Singleton( );

    /* A private Constructor prevents any other
     * class from instantiating.
     */
    private Singleton() { }

    /* Static 'instance' method */
    public static Singleton getInstance( ) {
        return singleton;
    }

    /* Other methods protected by singleton-ness */
    protected static void demoMethod( ) {
        System.out.println("demoMethod for singleton");
    }
}
```

prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

31

NOTES

1 The Java class:

CODE	WHAT IT DOES
public class Singleton {	
private static Singleton singleton = new Singleton();	The one instance, created when the class loads (eager initialisation).
private Singleton() { }	A private constructor prevents any other class from instantiating it.
public static Singleton getInstance() { return singleton; }	The static "instance" method: always returns the same object.
protected static void demoMethod() { System.out.println("demoMethod for singleton"); } }	Other methods, reached through the instance.

- 2 A **static** variable is allocated statically: it lives for the entire run of the program, which is why it can hold the single instance. Compare with the exam version (slide 27), which creates the instance lazily inside getInstance().

Java Code for the Singleton Pattern

```
// File Name: SingletonDemo.java
public class SingletonDemo {

    public static void main(String[] args) {
        Singleton tmp = Singleton.getInstance( );
        tmp.demoMethod( );
    }
}
```

The main program file where we will create a singleton object

NOTES

1 Using it:

CODE

```
public class SingletonDemo { public
static void main(String[] args) {
```

```
    Singleton tmp =
    Singleton.getInstance();
```

```
        tmp.demoMethod(); } }
```

WHAT IT DOES

The main program.

Not `new Singleton()` (that would not compile): ask the class for its instance.

Use it like any object.

Singleton Design Pattern Example

➤ Steps to design using the Singleton design pattern:

1. Set the visibility of the **constructor** to **private**
2. Define a **private static member** attribute that can store a reference (i.e., a pointer) to the one instance of the singleton.
3. Create a **public static getInstance()** method that can access the private constructor to instantiate one object of the singleton type and return it to clients.

NOTES

1. Make the **constructor private**.
2. Define a **private static** attribute that stores a reference to the one instance.
3. Create a **public static** `getInstance()` method that uses the private constructor to create one object (if it doesn't exist yet) and returns it to clients. These three steps are the whole answer to "write the Singleton code".

EXAM ANGLE

Prediction

Q1

List the three steps to design a class with the Singleton pattern.

Answer: (1) Make the constructor private. (2) Add a private static attribute holding the single instance. (3) Add a public static `getInstance()` that creates the instance through the private constructor if it doesn't exist and returns it.

Benefits of the Singleton Design Pattern

➤ Benefits:

- ✓ It provides controlled access to a single instance of a given type.
- ✓ It can be customized to permit a variable number of instances.

➤ Drawback:

- ✓ It does not work well in multithreading environments

NOTES

- 1 Benefit:** controlled access to a **single instance** of a type.
- 2 Benefit:** it can be customised to allow a **variable number** of instances (e.g. a pool of at most N), since creation goes through one method.
- 3 Drawback:** it doesn't work well in **multithreaded** environments: two threads calling `getInstance()` at the same moment can both see `instance == null` and create two objects, unless the method is synchronised.

EXAM ANGLE

Prediction

Q1

True or false: a Singleton's constructor should be public so the instance can be created when needed.

Answer: False — It must be private; clients get the instance through the public static `getInstance()`.

Prediction

Q2

Why can the lazy Singleton fail in a multithreaded program, and how is it fixed?

Answer: Two threads can call `getInstance()` at the same time, both find `instance == null` and both create an object. Fix: synchronise `getInstance()` (or use eager initialisation / double-checked locking).

Extra: Real-World Scenario

Let's say you're building a **Banking App**:

Login System logs when a user logs in.

Transaction System logs when money is transferred.

Notification System logs when a message is sent.

☞ You want all events to go to the same log, in the correct order.

Step 1: Define the Singleton Class EventLogger

```
java
public class EventLogger {
    private static EventLogger instance;

    // Private constructor to prevent outside creation
    private EventLogger() {}

    // Global access point
    public static EventLogger getInstance() {
        if (instance == null) {
            instance = new EventLogger();
        }
        return instance;
    }

    // Method to Log messages
    public void log(String message) {
        System.out.println("Log: " + message);
        // In real use: write to a file
    }
}
```

Step 2: Use Singleton in Different Parts of the App**Login System**

```
java
public class LoginSystem {
    public void login(String username) {
        // Do Login work...
        EventLogger.getInstance().log("User " + username + " logged in.");
    }
}
```

Why don't we write `instance.log()` directly?

Because **instance** is **private** and **static** inside the class, and we're **outside** the class when we want to use it.

- `getInstance()` is a **public static method** → you can access it from anywhere.
- It **returns the instance** (the one and only object).
- Then you can call `.log()` on that object.

35

NOTES

1 The scenario: a banking app's login, transaction and notification systems all log events, and everything must go to **one log in the correct order**.

2 Step 1, the Singleton class:

CODE**WHAT IT DOES**

```
public class EventLogger { private
static EventLogger instance;
```

The one instance.

```
private EventLogger() {}
```

Prevents outside creation.

```
public static EventLogger
getInstance() { if (instance == null)
{ instance = new EventLogger(); }
return instance; }
```

The global access point, created on first use.

```
public void log(String message) {
System.out.println("Log: " + message);
} }
```

In real use this would write to a file.

3 Step 2, the login system: `EventLogger.getInstance().log("User " + username + " logged in.");`. Why not `instance.log()`? Because `instance` is private and static inside the class; from outside, `getInstance()` is the public way to reach the one object, and then you call `.log()` on it.

Extra: Transaction System

```
public class TransactionSystem {
    public void transfer(String from, String to, double amount) {
        // Do transfer work...
        EventLogger.getInstance().log("Transferred $" + amount + " from " + from + " to " + to);
    }
}
```

Notification System

```
public class NotificationSystem {
    public void sendNotification(String user, String message) {
        // Do notification work...
        EventLogger.getInstance().log("Notification sent to " + user + ": " + message);
    }
}
```

Step 3: Main Class to Run the App

```
public class Main {
    public static void main(String[] args) {
        LoginSystem login = new LoginSystem();
        login.login("Alice");

        TransactionSystem transaction = new TransactionSystem();
        transaction.transfer("Alice", "Bob", 250.0);

        NotificationSystem notify = new NotificationSystem();
        notify.sendNotification("Alice", "Transfer completed");

        // Proof: All use the same Logger instance
        System.out.println("Logger check: " +
            (EventLogger.getInstance() == EventLogger.getInstance())); // true
    }
}
```

prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

NOTES

- 1 The **transaction system** logs "Transferred \$" + amount + " from " + from + " to " + to through `EventLogger.getInstance().log(...)`.
- 2 The **notification system** logs "Notification sent to " + user + ": " + message the same way.
- 3 The **main class** runs login, a transfer and a notification, then prints `EventLogger.getInstance() == EventLogger.getInstance()`, which is `true`: proof that all three systems used the same logger object.

Singleton Design Pattern Example

```
#include "EventManager.h"
#include <iostream>

using namespace std;
```

```
// Initialize the instance_ static member attribute.
EventManager* EventManager::_instance = 0;
```

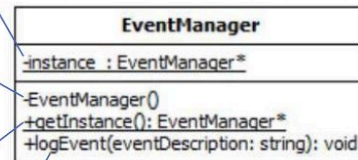
```
// Private constructor.
EventManager::EventManager(void)
{
    // Intentionally left blank.
}
```

```
// The global point of access to the EventManager.
EventManager* EventManager::getInstance(void) {

    // Determine if an instance of the EventManager has been created.
    if( _instance == 0 ) {

        // Create the one and only instance.
        _instance = new EventManager;
    }
    return _instance;
}
```

```
// The method that logs events.
void EventManager::logEvent(string eventDescription) {
    // Code to log event.
    cout << eventDescription << endl;
}
```



prince Sultan University | Slides prepared from C++ (Practical) textbook and slides

37

NOTES

1 The textbook EventManager in C++:

CODE

```
EventManager* EventManager::_instance
= 0;
```

WHAT IT DOES

Initialise the static member: no instance yet.

```
EventManager::EventManager(void) { /*
Intentionally left blank. */ }
```

The private constructor.

```
EventManager*
EventManager::getInstance(void) {
```

The global point of access.

```
    if ( _instance == 0 ) { _instance =
new EventManager; }
```

Create the one and only instance the first time.

```
    return _instance; }
```

Every call returns the same pointer.

```
void EventManager::logEvent(string
eventDescription) { cout <<
eventDescription << endl; }
```

The method that logs events.

Extra: Example A drawing application needs to create many copies of two types of shapes: **Circle** and **Square**.

Each shape has one attribute (Circle → radius, Square → side).

The system should allow creating new shapes **by copying existing ones** instead of creating them from scratch every time.

Tasks:

1. Draw a class diagram that solves this problem.
2. Write Java code that:
 1. Defines a common interface for all shapes.
 2. Allows each shape to make a copy of itself.
 3. Demonstrates copying a Circle and a Square in the `main` method.

(Hint for students: Think of a design pattern where objects can create duplicates of themselves.)

How Students Should See It

Step 1 → Prototype interface with `clone()`.

Step 2 → Concrete Prototype A (Circle).

Step 3 → Concrete Prototype B (Square).

Step 4 → Client (PrototypeDemo) that uses cloning instead of `new`.

NOTES

- 1 An extra exercise on the fifth creational pattern, **Prototype**: a drawing app needs many copies of **Circle** (radius) and **Square** (side).
- 2 New shapes should be created **by copying existing ones** instead of from scratch.
- 3 Tasks: draw the class diagram; write Java with a common interface, a self-copy method in each shape, and a `main` that copies a Circle and a Square.
- 4 The plan: a Prototype interface with `clone()`, concrete prototypes Circle and Square, and a client that clones instead of calling `new`.

EXAM ANGLE

Prediction

Q1

Which pattern creates new objects by copying an existing instance?

- A. Prototype
- B. Builder
- C. Singleton
- D. Composite

Answer: A. Prototype

Step 1: Define the common interface

java

// Step 1: Interface for all shapes

```
interface Shape {  
    Shape clone();    // method to duplicate the object  
}
```

NOTES

1 Step 1:

CODE

```
interface Shape { Shape clone(); }
```

WHAT IT DOES

Every shape can duplicate itself; the copy is returned as a Shape.

Step 2: Create the first concrete class (Circle)

```
java

// Step 2: Concrete class - Circle
class Circle implements Shape {
    int radius;

    Circle(int r) { this.radius = r; }

    @Override
    public Shape clone() {
        return new Circle(this.radius); // make a copy of Circle
    }

    public void show() {
        System.out.println("Circle with radius " + radius);
    }
}
```

42

NOTES**1** Step 2:**CODE**

```
class Circle implements Shape { int
radius; Circle(int r) { this.radius =
r; }
```

WHAT IT DOES

A concrete prototype with its state.

```
@Override public Shape clone() {
return new Circle(this.radius); }
```

Makes a copy with the same radius.

```
public void show() {
System.out.println("Circle with radius
" + radius); } }
```

Step 3: Create the second concrete class (Square)

```
java

// Step 3: Concrete class - Square
class Square implements Shape {
    int side;

    Square(int s) { this.side = s; }

    @Override
    public Shape clone() {
        return new Square(this.side); // make a copy of Square
    }

    public void show() {
        System.out.println("Square with side " + side);
    }
}
```

NOTES**1** Step 3:**CODE**

```
class Square implements Shape { int
side; Square(int s) { this.side = s; }
```

```
    @Override public Shape clone() {
return new Square(this.side); }
```

```
    public void show() {
System.out.println("Square with side "
+ side); } }
```

WHAT IT DOES

The second concrete prototype.

Copies the side length.

Step 4: Write the client (main method)

```

java

// Step 4: Client code to test cloning
public class PrototypeDemo {
    public static void main(String[] args) {
        Shape c1 = new Circle(10); // create original Circle
        Shape c2 = c1.clone();      // clone it

        Shape s1 = new Square(5);  // create original Square
        Shape s2 = s1.clone();      // clone it

        ((Circle)c1).show();
        ((Circle)c2).show();
        ((Square)s1).show();
        ((Square)s2).show();
    }
}

```

Expected Output:

```

pgsql

Circle with radius: 10
Circle with radius: 10
Square with side: 5
Square with side: 5

```

Key Idea:

- Clone creates identical copy
- So:
 - `c1 = c2`
 - `s1 = s2`

NOTES**1** Step 4, the client:**CODE**

```
Shape c1 = new Circle(10); Shape c2 = c1.clone();
```

WHAT IT DOES

Create an original circle, then clone it.

```
Shape s1 = new Square(5); Shape s2 = s1.clone();
```

The same for a square.

```
((Circle)c1).show();
((Circle)c2).show();
((Square)s1).show();
((Square)s2).show();
```

Prints "Circle with radius 10" twice and "Square with side 5" twice.

- 2** Key idea: a clone is an identical but separate copy (c1 and c2 have equal state but are different objects). With nested mutable fields, decide between shallow and deep copies.

Summary

- In this session, we continued the discussion on **creational** design patterns, including:
 - ✓ **Builder**
 - ✓ **Singleton**
- Next ... we will present **structural** and **behavioral** design patterns in detailed design.

NOTES

- 1 This lecture: **Builder** and **Singleton** (plus a Prototype exercise), completing the creational patterns.
- 2 Next: **structural** and **behavioural** design patterns (Chapter 7).

Original slides from the SE322 lecture deck, with study notes. Check exact wording against the [original deck](#).
Notes and questions are study aids. Made by Shoug Alomran.